FACULTATEA: Automatică și Calculatoare
SPECIALIZAREA: Calculatoare și Tehnologia Informației
DISCIPLINA: Proiectarea sistemelor numerice
PROIECT: MEMORIE FIFO

Îndrumător laborator

Csillag Szabolcs-Andras

Realizat de

Creț Maria-Magdalena

Grupa: 30213, Semigrupa 1

CONTENTS

Specificații.....3

Black box memorie FIFO.....4

Componente.....4-6

Codul VHDL.....7-13

Organigrama memoriei FIFO.....11

Schema în detaliu a implementării memoriei FIFO.....14

Motivarea alegerii implementării.....14

Dezvoltări ulterioare.....14

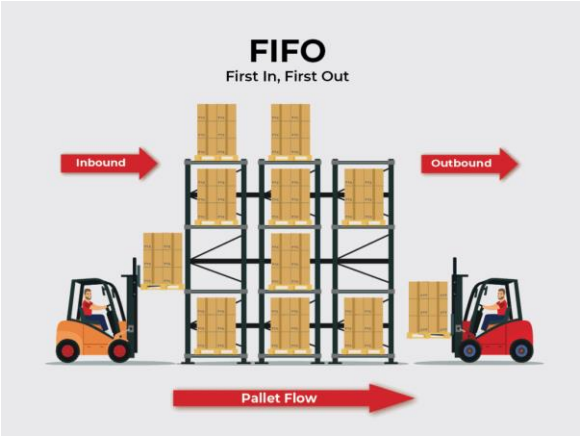
Bibliografie.....15

1. Specificații

CERINȚĂ: Să se implementeze schemele de construire a memoriilor FIFO, conform documentației existente în cartea *"Proiectarea sistemelor numerice folosind tehnologia FPGA"* de Sergiu Nedevschi, Zoltan Baruch și Octavian Creț în capitolul 4 al lucrării.

Memoria FIFO are la bază logica *"**F**irst **I**n, **F**irst **O**ut"* întâlnită și în alte limbaje de progamare, sub denumirea de coadă. Informația introdusă prima dată în memorie este și prima care va fi afișată, fiind stocată într-o adresă a memoriei RAM, afișată atunci când utilizatorul dorește prin apăsarea butonului POP.

Folosind facilități oferite de memoria RAM disponibilă în cadrul cipului FPGA, proiectele bazate pe memorii pot opera la viteze mult mai mari. Modul de lucru cu port dual permite scrierea și citirea simultană a datelor, cee ace multiplică de patru ori viteza memoriei FIFO față de implementările cu memorii asincrone.



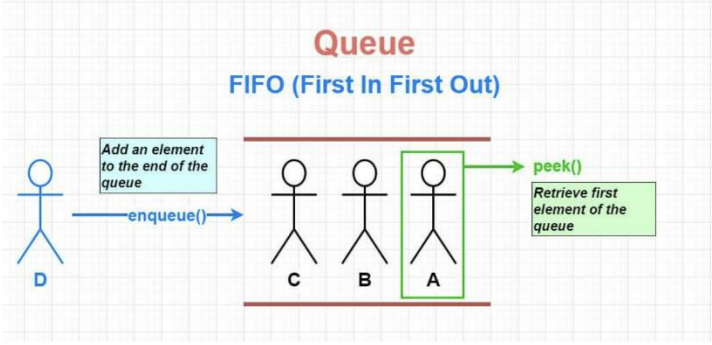


Figura 1.1 și Figura 1.2 *Logica memoriei FIFO*

PROIECTARE

2. Black box-ul Memoriei FIFO implementată

O memorie FIFO clasică constă în patru blocuri logice separate, după cum se poate vedea în figura 2.1. Blocul de control generează semnalele de validare ale scrierii și citirii. Logica de stare semnalează dacă memoria FIFO este plină sau vidă, sau poate semnala orice altă stare, în cazul nostru ”plin minus unu”. Pointerii de adrese de scriere și citire sunt stocați în două număratoare directe, iar datele sunt stocate într-o memorie RAM. Logica de control și de stare sunt în strânsă dependență de aplicație.

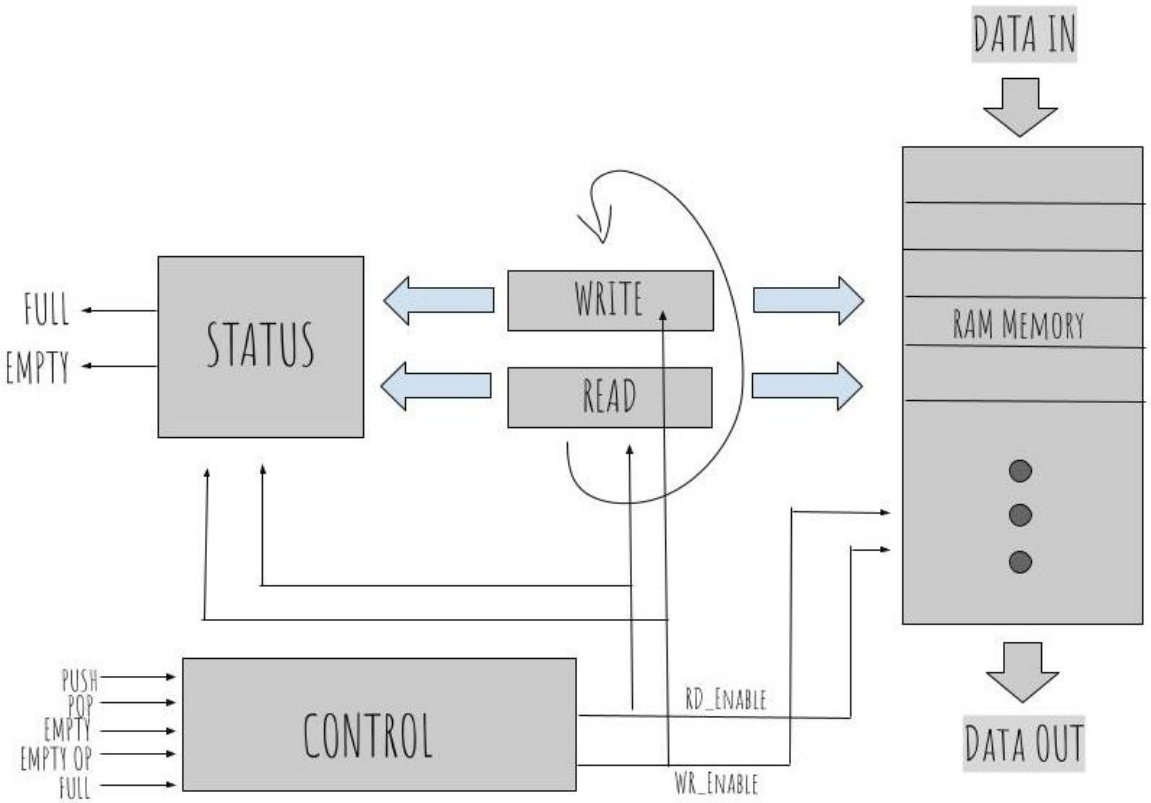


Figura 2.1 *Diagrama bloc a memoriei FIFO*

3. Componente

1. Componenta de control sau *unit_command* modifică intrările PUSH și POP în mai multe semnale, dintre care cel mai important este *write*, odată cu citirea numărului adăugat în memorie în limbaj binar, de biți 1 și 0, respectiv translatarea lor în limbaj zecimal, cu ajutorul convertorului, acesta va fi și afișat pe afișorul BCD 7 segmente. Aceste semnale sunt folosite și în celelalte componente principale memoriei FIFO. În cadrul arhitecturii se găsește un proces care are semnalele PUSH, POP, EMPTY, EMPTY_OP (care realizeaza operația de golire a memoriei manuală, atunci când utilizatorul dorește să golească întreaga memorie instant), FULL și un semnal de CLK (clock), care funcționează concomitent și în cazul numărătoarelor integrate în implementarea memoriei, cât și în cazul memoriei RAM. Când unul dintre acestea se modifică, procesul se va relua.

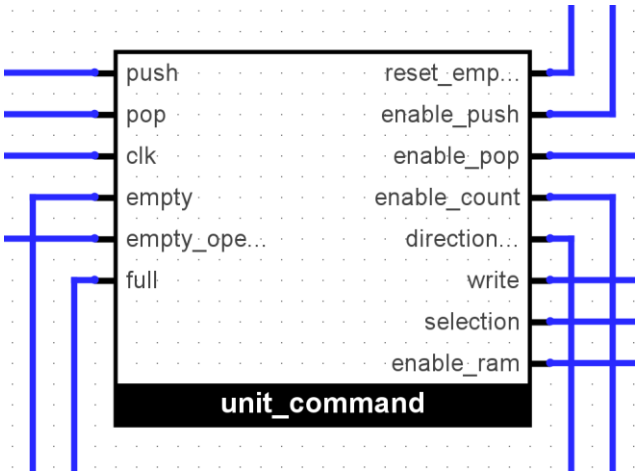


Figura 3.1 *unit_command (control component)*

Ieșiri :

Reset_empty_operation – ieșire care va reseta memoria și toate celelalte numărătoare
Enable_push – ieșire care va activa enable pentru numărătorul operației PUSH
Enable_pop– ieșire care va activa enable pentru numărătorul operației POP
Enable_count– ieșire care va activa enable pentru numărătorul operației de numărare a elementelor
Direction_count – ieșire care va decide modul de numărare a elementelor (crescător sau descrescător)
Write – ieșire care va activa operația de scriere în memoria RAM
Selection – ieșire care va activa selecția mux-ului 2:1
Enable_ram – ieșire care va activa enable pentru memoria RAM.

2. Componenta de status sau *status_component* are rolul principal de a avertiza utilizatorul de stadiul memoriei FIFO, ca fiind goala (EMPTY) sau plina (FULL). Acest lucru este realizat cu ajutorul unui numărător reversibil direct. De asemenea, semnalele EMPTY sau FULL sunt transmise vizual cu ajutorul unor leduri. Intrarea componentei status o reprezintă numărul de elemente realizat de numărătorul implementat. Când numărătorul se află în starea 00000, aceasta activează ieșirea EMPTY, deoarece memoria este goală.

Când numărătorul este în starea 11111 (32 în zecimal), aceasta activează ieșirea FULL, deoarece memoria este plină.

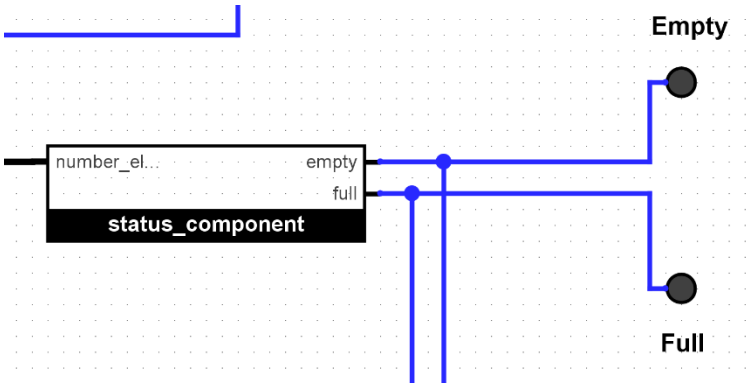


Figura 3.2 *status component*

3. Numărătorul reversibil sincron este utilizat atât pentru a număra în cazul operațiilor PUSH și POP cât și pentru a reține numărul de elemente care se află în memorie. Rolul său în ceea ce consta statusul memoriei, EMPTY sau FULL a fost menționat la componenta anterioară.
   Operații: enable = 1, numărătorul este activ
   direction = 1, se numără crescător
   direction = 0, se numără descrascător
   reset = 1, se resetează numărarea

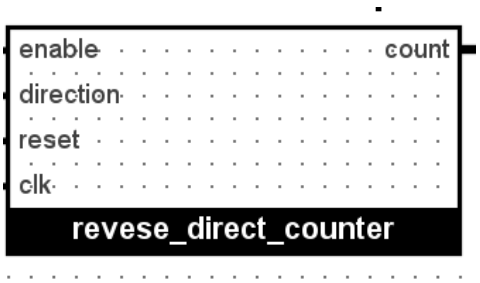


Figura 3.3 *reverse direct counter*

4. Multiplexorul cu 2 intrări, respectiv o ieșire, pentru numere pe 5 biți.

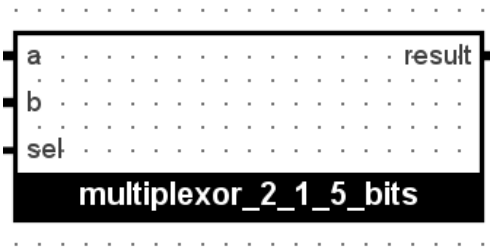


Figura 3.4 *multiplexor 2-1, pe o cale de 5 biți*

5. Convertor binar în zecimal are rolul de a converti numerele adăugate în binar pe 8 biți, în limbaj zecimal, atât unitățiile, cât și zecile și sutele.

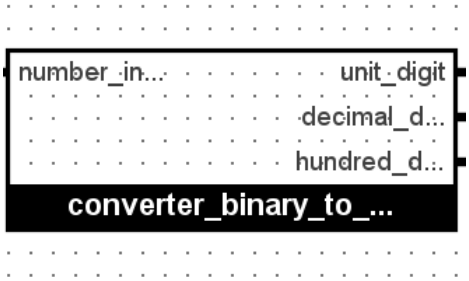


Figura 3.5 *converter binary to decimal*

6. Memoria RAM este o memorie sincrona 32x8, 32 de adrese , iar cuvintele sunt pe 8 biți. Operațiile memoriei se efectuează doar pe frontal crescător al clock-ului și dacă enable = 1. Dacă reset este activ atunci cuvântul de la adresa dată va fi 0. Scrierea se va efectua doar daca w_r = 1 (write-read), iar pe ieșire se va afișa ultimul element scris în memorie sau când w_r = 0, elementul citit de la adresa data.

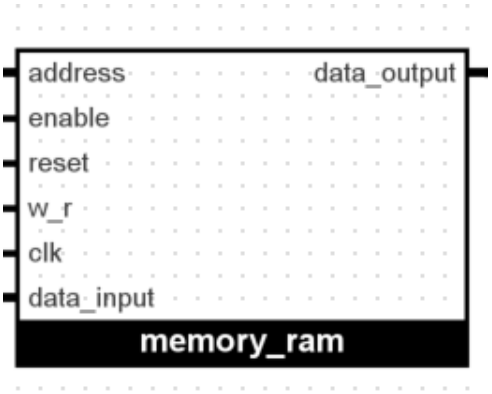


Figura 3.6 *memory RAM*

4. Codul VHDL

Proiectul a fost implementat cu ajutorul codului VHDL, care a apărut în anul 1987 și se află în forma actuală încă din anul 1993. Este utilizat în concepția asistată pe calculator (CAD=Computer Aided Design), a circuitelor integrare (de exemplu ASIC) sau, în cazul nostru, pentru configurarea FPGA-urilor. Codul VHDL care sta la baza implementării memoriei FIFO este alcătuit din 6 secțiuni: *converter_binary_to_decimal*, *memory_ram*, *reverse_direct_counter*, *multiplexor_2_1_5_biți*, *unit_command* și *status_component*, fiecare dintre acestea realizând câte o componentă care stă la baza implementării.

În cazul acestui proiect s-au folosit 4 biblioteci din librăria IEEE, în special în secțiunea de cod care realizează convertirea din binar în zecimal, unde au fost utilizate toate cele patru. Acestea sunt: **`ieee.std_logic_1164.all`** (definește tipurile de date pentru a lucre cu semnale logice, cum ar fi std_logic si std_logic_vector), **`ieee.std_logic_unsigned.all`** (pentru a putea folosi standard logic și nu termenul de bit, permite operatii matematice cu semnale logice), **`ieee.std_logic_arith.all`** (definește tipurile de date și funcțiile necesare pentru a lucre cu numere întregi și numere în virgula mobilă în VHDL) , **`ieee.numeric_std.all`** (furniceaza funcții matematice pentru a putea efectua diferite operații asupra valorilor numerice).

Fiecare componentă este construită separat, cu ajutorul proceselor sau în mod structural, prin conectarea cu alte componente.

Componenta *converter_binary_to_decimal* realizează convertirea din binar în zecimal pentru a putea afișa pe BCD 7 segmente numărul introdus prin pini de 0 și 1 (8 pini, intrări), în limbaj zecimal, astfel că am utilizat un BCD 7 segmente pentru cifra unităților reprezentată pe un număr format din 3 biți și analog pentru cifra zecilor și cea a sutelor. S-au utlizat toate cele patru biblioteci menționate mai sus. Codul acestei componente este format din secțiunea de declarare a variabilelor utilizate în implementare, din inițializarea unor semnale, din arhitectura utilizată, urmată de secțiunea de cod care realizeaza acest convertor.

```
--------------------------------------------------------------------------------
-- Project : Memorii FIFO
-- Autor   : Cret Maria-Magdalena
--------------------------------------------------------------------------------
-- Description : Implementarea unui convertor din binar in zecimal
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
USE ieee.numeric_std.all;

ENTITY converter_binary_to_decimal IS
  PORT (
  number_in_digit: in std_logic_vector(7 downto 0);
  unit_digit: out std_logic_vector(3 downto 0);
  decimal_digit: out std_logic_vector(3 downto 0);
  hundred_digit: out std_logic_vector(3 downto 0)
    );
END converter_binary_to_decimal;


ARCHITECTURE TypeArchitecture OF converter_binary_to_decimal IS
signal number_decimal, unit_decimal, decimal_decimal, hundred_decimal : integer;
BEGIN
     number_decimal<= conv_integer(number_in_digit);
     unit_decimal<=number_decimal rem 10;
     decimal_decimal<=(number_decimal rem 100)/10;
     hundred_decimal<=number_decimal/100;

     process(unit_decimal, decimal_decimal, hundred_decimal)
     begin
          unit_digit<=std_logic_vector(to_unsigned(unit_decimal, 4));
          decimal_digit<=std_logic_vector(to_unsigned(decimal_decimal, 4));
          hundred_digit<=std_logic_vector(to_unsigned(hundred_decimal, 4));
     end process;

END TypeArchitecture;
```

Figura 4.1 *Codul componentei converter_binary_to_decima*

Componenta *memory_ram* reprezinta o memorie RAM, implementată cu ajutorul librăriilor **`ieee.std_logic_1164.all`** și **`ieee.numeric_std.all`**, sunt inițializate doua semnale, *memory_signal* și *address_aux,* iar memoria este una de 32x8 biți. Codul care stă în implementarea acesteia este următorul:

```
----------------------------------------------------------------------------
-- Project : Memorii FIFO
-- Autor   : Cret Maria-Magdalena
----------------------------------------------------------------------------
-- Description : implementarea memoriei RAM
----------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY memory_ram IS
  PORT (
     address: in std_logic_vector(4 downto 0);
     enable: in std_logic;
     reset: in std_logic;
     w_r: in std_logic;
     clk: in std_logic;
     data_input: in std_logic_vector(7 downto 0);
     data_output: out std_logic_vector(7 downto 0)
    );
END memory_ram;

ARCHITECTURE TypeArchitecture OF memory_ram IS
type memory_32_8 is array (0 to 31) of std_logic_vector(7 downto 0);
signal memory_signal: memory_32_8:=(others=> x"00");
signal adress_aux: std_logic_vector(4 downto 0):="00000";
BEGIN
     process(clk, address, w_r, data_input)
     begin
          if clk'event and clk='1' then
               if reset='1' then
                    memory_signal<=(others=> x"00");
               elsif enable='1' then
                    if w_r='1' then
                         memory_signal(to_integer(unsigned(address)))<=data_input;      -- ia elementul dim memorie de la adresa data
                         adress_aux<=address;
                    else adress_aux<=address;
                    end if;
               end if;
          end if;
     end process;

     data_output<=memory_signal(to_integer(unsigned(adress_aux)));

END TypeArchitecture;
```

Figura 4.2 *Codul componentei memory_ram*

Componenta *reverse_direct_counter* reprezintă un numărător reversibil sincron, care este utilizat în implementarea memoriei FIFO de 3 ori, atat pentru Push-Up, de câte ori se realizează scrierea unui număr în memorie, cât și pentru Pop-Up, de câte ori se realizează citirea unui număr din memorie și pentru a număra câte elemente se afla în memorie la fiecare pas. Codul care stă în implementarea acestuia este următorul:

```
----------------------------------------------------------------------------------
-- Project : Memorii FIFO
-- Autor   : Cret Maria-Magdalena
----------------------------------------------------------------------------------
-- Description : Implementarea unui numarator direct reversibil pe care il vom utiliza in implementarea memoriei FIFO
----------------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;

ENTITY revese_direct_counter IS
  PORT (
     enable: in std_logic;
     direction: in std_logic;
     reset: in std_logic;
     clk: in std_logic;
     count: out std_logic_vector(4 downto 0)
    );
END revese_direct_counter;


ARCHITECTURE TypeArchitecture OF revese_direct_counter IS
signal count_aux: std_logic_vector(4 downto 0):= (others=> '0');
BEGIN
     process(clk, direction, enable, reset)
     begin
          if clk'event and clk='1' then
               if reset='1' then
                    count_aux<="00000";
               elsif enable='1' then
                    if direction='1' then
                         count_aux<= count_aux+"00001";
                    else
                         count_aux<= count_aux-"00001";
                    end if;
               end if;
          end if;
     count<=count_aux;
     end process;

END TypeArchitecture;
```

Figura 4.2 *Codul componentei reverse_direct_counter*

Componenta *multiplexor_2_1_5_biți* reprezintă un multiplexor c u 2 intrări și o ieșire, pe 5 biți, intrarea a reprezentânfd ieșitrea din numărătorul Push-Up, iar intrarea b, ieșirea pentru numărătorul Pop-Up, alegând astfel în funcție de selecție care dintre acestea să fie adăugat în memoria RAM. SEL = 0, se adaugă a, altfel se adaugă b. Codul care stă în implementarea acestuia este următorul:

```
----------------------------------------------------------------------------------
-- Project : Mmeorii FIFO
-- Autor   : Cret Maria-Magdalena
----------------------------------------------------------------------------------
-- Description : Implementarea unui multiplexor cu 2 intrari si o iesire pe o cale de 5 biti
----------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY multiplexor_2_1_5_bits IS
  PORT (
     a: in std_logic_vector(4 downto 0);
     b: in std_logic_vector(4 downto 0);
     sel: in std_logic;
     result: out std_logic_vector(4 downto 0)
    );
END multiplexor_2_1_5_bits;

ARCHITECTURE TypeArchitecture OF multiplexor_2_1_5_bits IS

BEGIN
     process(a, b, sel)
     begin
          if sel='0' then
               result<=a;
          else
               result<=b;
          end if;
     end process;

END TypeArchitecture;
```

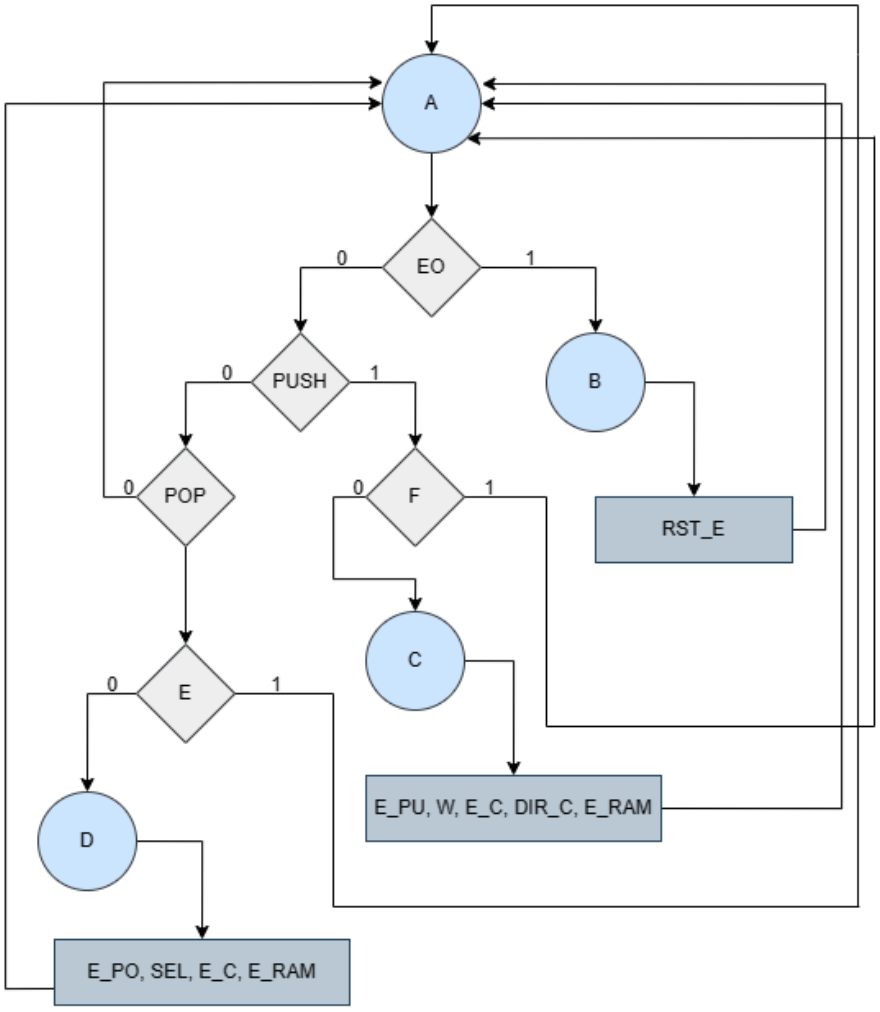
Figura 4.3 *Codul componentei multiplexor_2_1_5_biți*

In cadrul componentei *status_component* se verifică dacă memoria este goală sau plină în funcție de numărul de elemente. Codul care stă la baza implementării acestei componente este următorul:

```
----------------------------------------------------------------------------------
-- Project : Memorii FIFO
-- Autor   : Cret Maria-Magdalena
----------------------------------------------------------------------------------
-- Description : Implementarea componentei status
----------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY status_component IS
  PORT (
     number_elements: in std_logic_vector(4 downto 0);
     empty: out std_logic;
     full: out std_logic
    );
END status_component;

ARCHITECTURE TypeArchitecture OF status_component IS

BEGIN
     process(number_elements)
     begin
     empty<='0';
     full<='0';
          if(number_elements="00000") then
               empty<='1';
          elsif(number_elements="11111") then
               full<='1';
          end if;
     end process;

END TypeArchitecture;
```

Figura 4.4 *Codul componentei status_component*

Pentru *unit_command* s-a implementat un automat de stare pe baza organigramei de jos:



C - PUSH STATE
D - POP STATE
A - WAIT STATE
B - EMPTY OPERATION STATE
E - EMPTY
F - FULL
E-O - EMPTY OPERATION

Figura 4.5 *Organigrama memoriei FIFO*

Codul care stă la baza implementării unității de comandă este următorul:

```
----------------------------------------------------------------------------------
-- Project : Memorii FIFO
-- Autor   : Cret Maria-Magdalena
----------------------------------------------------------------------------------
-- Description : Implementarea unitatii de comanda a intregii memorii
----------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY unit_command IS
  PORT (
  push: in std_logic;
  pop: in std_logic;
  clk: in std_logic;
  empty: in std_logic;
  empty_operation: in std_logic;
  full: in std_logic;

  reset_empty_operation:out std_logic;
  enable_push: out std_logic;
  enable_pop: out std_logic;
  enable_count: out std_logic;
  direction_count: out std_logic;
  write: out std_logic;
  selection:out std_logic;
  enable_ram: out std_logic
    );
END unit_command;

ARCHITECTURE TypeArchitecture OF unit_command IS
type state_fifo is (Wait_state, Empty_operation_state, Push_state, Pop_state);
signal state, next_state: state_fifo:=Wait_state;
BEGIN

     --- Procesul care trece de la o stare la alta
     process(clk)
     begin
          if clk'event and clk = '1' then
               state <= next_state;
          end if;
     end process;
```

Figura 4.6 *Codul componentei unit_command – prima parte*

```
    --- Procesul care decide noua stare pe baza vechii starii si a intrariilor
    process(state, push, pop, empty, full, empty_operation)
    begin
    case state is
         when Wait_state => if empty_operation='1' then next_state<=Empty_operation_state;
                            elsif push='1' then
                                 if full='0' then
                                   next_state<=Push_state;
                                 else
                                   next_state<=Wait_state;
                                 end if;
                            elsif pop='1' then next_state<=Pop_state;
                                 if empty='0' then
                                   next_state<=Pop_state;
                                 else
                                   next_state<=Wait_state;
                                 end if;
                            else next_state<=Wait_state;
                            end if;
         when Empty_operation_state => next_state<=Wait_state;
         when Push_state => next_state<=Wait_state;
         when Pop_state => next_state<=Wait_state;
    end case;
    end process;

    --- Procesul care activeaza iesirile in functie de stari
    process(state, push, pop, empty, full, empty_operation)
    begin

    reset_empty_operation<='0';
    enable_push<='0';
    enable_pop<='0';
    enable_count<='0';
    direction_count<='0';
    write<='0';
    selection<='0';
    enable_ram<='0';

    case state is
         when Empty_operation_state => reset_empty_operation<='1';
         when Push_state => enable_push<='1'; write<='1'; enable_count<='1'; direction_count<='1';enable_ram<='1';
         when Pop_state => enable_pop<='1'; selection<='1';enable_count<='1';enable_ram<='1';
         when others => selection<='0';
    end case;
    end process;

END TypeArchitecture;
```

Figura 4.7 *Codul componentei unit_command – a doua parte*

5. Schema în detaliu a implementării memoriei FIFO

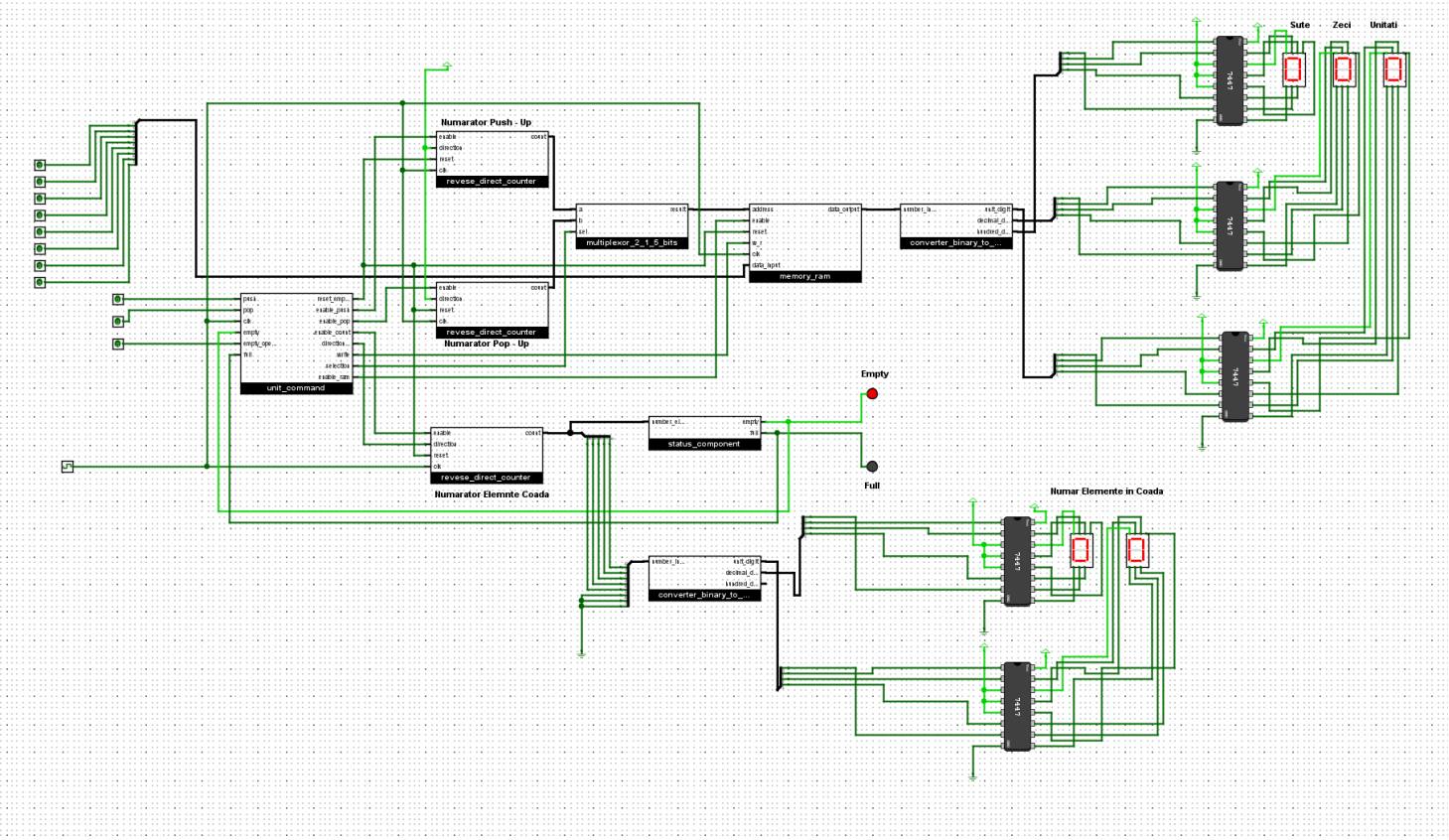


Figura 5.1 *Schema detaliată a proiectului conform implementării în Logisim*

6. De ce s-a ales această implementare a memoriei FIFO?

S-a implementat organigrama pentru a observa modul în care intrările sistemului influențează ieșirile acestuia. Pe baza acesteia s-a construit unitatea de comanda, iar pe baza unității de comandă, s-a realizat unitatea de execuție. Unitatea de execuție constă într-o memorie RAM 32x8, s-a ales această dimensiune pentru simplitatea modelului, numărătoare reversibile sincrone, care sunt folosite pentru operațiile de numărare push, respectiv pop. S-a ales folosirea aceluiași tip de numărător pentru a simplifica implementarea memoriei FIFO. Afișoarele BCD și convertoarele binar în zecimal sunt cele clasice pentru a putea vizualiza datele. S-a utilizat câte un afișor BCD 7 segmente pentru fiecare dintre uniăți, zeci, sute, cât și pentru numărarea elementelor aflate în memoria FIFO.

7. Dezvoltări ulterioare

Memoria FIFO poate fi dezvoltată treptat, prin mărirea capacității memoriei RAM, astfel încât să se poată introduce mai mult de 32 de cuvinte pe 8 biți. Se mai poate îmbunătății și prin introducerea unor noi funcționalități, de exemplu apariția unui led auxiliar atunci când ai ajuns cu citirea la cel din urmă element al memoriei. Implementarea unor numărătoare care reține numărul numerelor impare/pare/prime introduse prin comanda PUSH, introducerea unor noi comenzi, de exemplu comanda de golire doar a jumătății din coadă (memorie FIFO). Pot fi aduse diverse modificări și/sau îmbunătățiri.

Bibliografie

1. **Sergiu Nedevschi, Zoltan Baruch, Octavian Creț** -*Proiectarea sistemelor numerice folosind tehnologia FPGA*